Scrabble®

Entrega 2
42.5
Versión 1.2

Alexandre Vinent Padrol (alexandre.vinent)
Soukaïna Mahboub Mehboub (soukaina.mahboub)

Índice

1. Funcionalidades implementadas en el sistema

Autenticación

El sistema permite a los usuarios iniciar sesión con sus credenciales o registrarse como nuevos miembros. Toda la información, como el nombre de usuario, contraseña, puntuación y avatares, se guarda en archivos JSON para mantener la persistencia y facilitar futuras validaciones. También se permite cerrar la sesión mediante el menú lateral.

Gestión de Cuenta

Desde el panel de cuenta, el usuario puede:

- Cambiar su nombre de usuario y contraseña.
- Asignar o actualizar una foto de perfil.
- Eliminar su cuenta de forma definitiva, borrando todos sus datos asociados, como partidas jugadas, imágenes, etc.

Gestión de Partidas

Creación de Partida

El usuario selecciona el modo de juego (SOLO o DOS JUGADORES), el recurso (combinación de diccionario y bolsa) y asigna un identificador único. En partidas de dos jugadores, se solicita además el inicio de sesión o registro del segundo jugador antes de comenzar.

Carga de Partida

- **Última partida**: con un botón se recupera el estado más reciente de juego.
- **Partidas anteriores**: se muestran en una lista filtrada por recursos válidos y por la existencia del oponente. Desde esa vista también pueden reanudarse y eliminar partidas guardadas, seleccionandolas de la lista.

Interfaz de Juego

Se reproduce el tablero de Scrabble (15×15) con todas sus bonificaciones y el atril de letras del jugador y sus puntuaciones en el momento durante la partida. Las acciones disponibles incluyen:

- Poner o quitar fichas del tablero del atril del jugador.
- Pasar turno.

- RESET para cambiar las fichas del atril seleccionadas por el jugador por otras aleatorias, y añadiendo las fichas descartadas a la bolsa.

- Finalizar el turno tras colocar una palabra.

- La opción “Ayuda”: sugiere una palabra válida basada en las fichas disponibles en el atril del jugador y se colocan las fichas directamente en el tablero y se pasa turno al siguiente jugador.

- Un botón de salida ofrece dos opciones: abandonar la partida, finalizandola permanentemente, o salir guardando para retomar más tarde.

Tras cada jugada, las palabras se validan automáticamente y se asignan los puntos correspondientes.

Notificaciones y Guardado

- La interfaz muestra etiquetas en tiempo real que informan sobre errores, validaciones y confirmaciones de acción.

- El sistema realiza guardados automáticos tras cada turno.

Ranking de Jugadores

Presenta un listado ordenado de forma descendente según la mejor puntuación registrada. Solo aparecen usuarios activos con puntuación mayor que cero.

Gestión de Recursos

Los recursos (cada uno constituido por un **diccionario** y una **bolsa** de fichas) se pueden crear o editar de manera manual o mediante importación de archivos localmente o también se pueden eliminar. Se aplican validaciones estrictas de formato:

- Diccionario en mayúsculas, sin acentos y ordenado lexicográficamente.
- Bolsa con letra, cantidad y puntuación correctamente especificadas según normativa.

Manual de Usuario

Incluye un visor HTML sencillo para consultar el manual de uso, con formato claro y accesible.

Menú Lateral

Un panel desplegable accesible en todo momento que agrupa accesos rápidos a: Guardar partida, Manual, Ranking, Recursos, Gestión de cuenta y Salir. El usuario puede abrirlo o cerrarlo según su preferencia.

2. Capa de Dominio

2.1. Diagrama de la Capa de Dominio

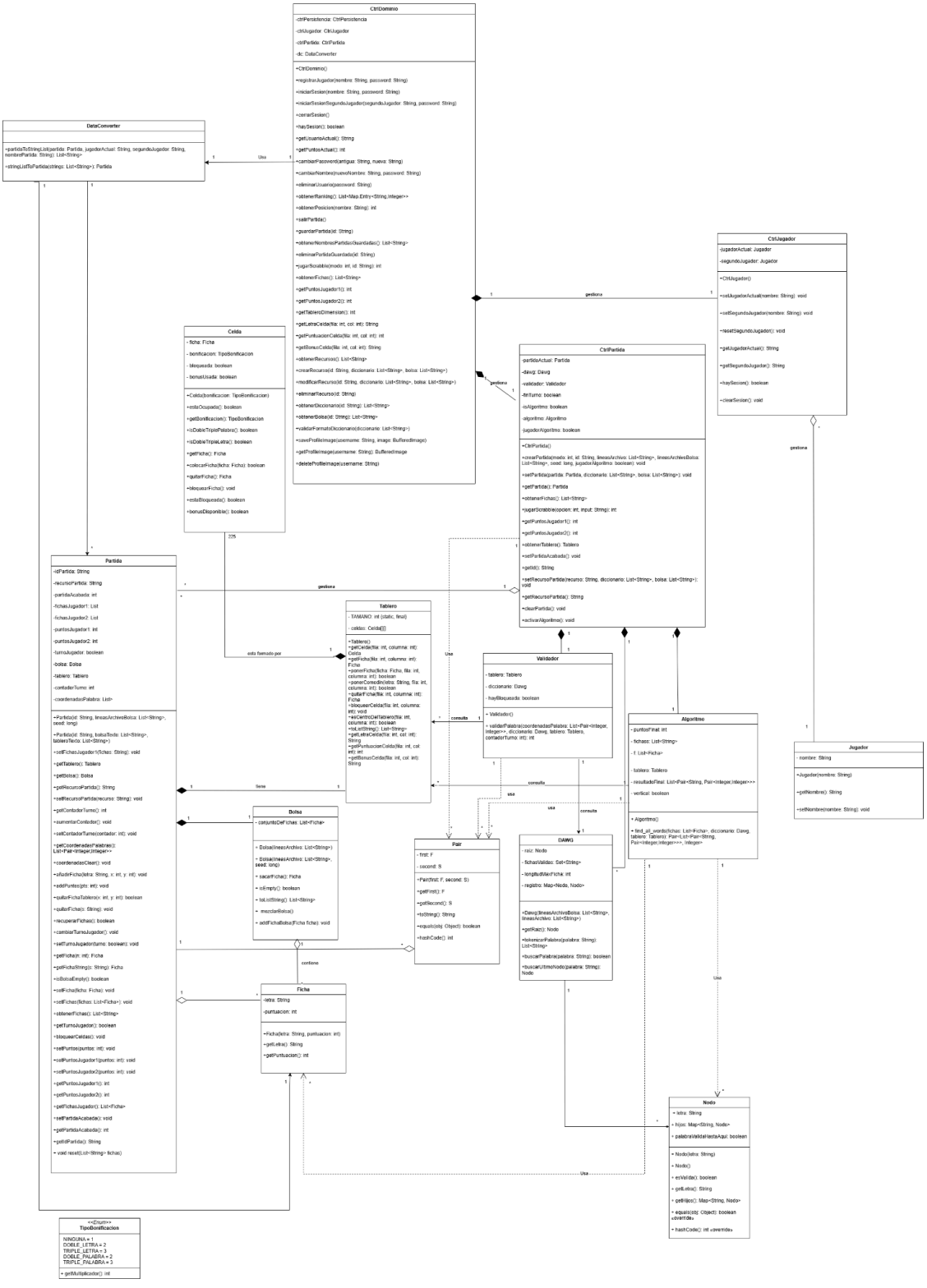

fig. 1 Diagrama de la Capa Dominio, del sistema implementado (Se encuentra en la carpeta DOCS).

2.2. Modelos

Algoritmo

Breve descripción:
Clase encargada de buscar y construir jugadas posibles en el juego de Scrabble, utilizando las fichas del jugador, el tablero y un diccionario DAWG para formar palabras válidas que respeten las reglas del juego. También calcula la puntuación de cada jugada y selecciona la mejor jugada encontrada.

Descripción de los atributos:

- **puntosFinal** (int): Puntuación total de la mejor jugada encontrada por el algoritmo.

- **fichass** (List<String>): Lista de letras disponibles en la mano del jugador (incluyendo comodines) que el algoritmo puede usar para formar palabras.

- **f** (List<Ficha>): Lista de objetos Ficha en la mano del jugador utilizadas en la búsqueda de palabras.

- **tablero** (Tablero): Referencia al tablero de juego actual donde se colocarán las palabras.

- **resultadoFinal** (List<Pair<String, Pair<Integer,Integer>>>): Jugada óptima encontrada, representada como una lista de pares de letra y posición que conforman la mejor palabra.

- **vertical** (boolean): Indicador de orientación para la búsqueda de palabras; determina si la palabra actual se construye en vertical (true) u horizontal (false).

Descripción de los métodos:

- **Algoritmo()**
  Constructor por defecto. Inicializa una instancia de Algoritmo sin realizar acciones adicionales.

- **Pair<List<Pair<String, Pair<Integer,Integer>>>, Integer> find_all_words(List<Ficha> fichas, Dawg diccionario, Tablero tablero)**
  Encuentra la mejor jugada posible en el tablero dadas las fichas del jugador. Evalúa todas las combinaciones válidas de palabras (horizontales y verticales) partiendo de las posiciones ancla, valida con el DAWG y calcula la puntuación total según bonificaciones. Retorna la jugada óptima y su puntuación.

Bolsa

Breve descripción:
Representa la bolsa de fichas del juego de Scrabble. La bolsa se puede inicializar a partir de configuraciones (por ejemplo, leídas de un fichero) con las letras, sus cantidades y puntuaciones; permite barajar aleatoriamente las fichas y extraer fichas de forma controlada durante la partida.

Descripción de los atributos:

- **conjuntoDeFichas** (List<Ficha>): Conjunto de todas las fichas actualmente disponibles en la bolsa.

Descripción de los métodos:

- **Bolsa(List<String> lineasArchivo)**
  Constructor que crea una nueva bolsa a partir de líneas de configuración en formato "Letra Cantidad Puntuacion". Añade las fichas correspondientes y mezcla la bolsa aleatoriamente.

- **Bolsa(List<String> lineasArchivo, long seed)**
  Igual que el anterior, pero usa una semilla (seed) para el mezclado aleatorio, permitiendo reproducibilidad.

- **Ficha sacarFicha()**
  Extrae y elimina la primera ficha de la bolsa. Devuelve la ficha extraída, o null si la bolsa está vacía.

- **boolean isEmpty()**
  Indica si la bolsa está vacía. Devuelve true si no quedan fichas; false en caso contrario.

- **List<String> toListString()**
  Devuelve una lista de cadenas que resumen las fichas restantes en la bolsa en formato "Letra Cantidad Puntuación".

- **void addFichaBolsa(Ficha ficha)**
  Añade una ficha a la bolsa.

- **void mezclarBolsa()**
  Mezcla las fichas de la bolsa aleatoriamente.

Celda

Breve descripción:
Representa una celda individual del tablero de Scrabble. Puede contener una ficha y una bonificación de letra o palabra, y mantiene información sobre si la ficha está bloqueada y si la bonificación ya fue usada.

Descripción de los atributos:

- **ficha** (Ficha): Ficha colocada en la celda (o null si está vacía).
- **bonificacion** (TipoBonificacion): Tipo de bonificación asignada a la celda (o NINGUNA).
- **bloqueada** (boolean): Indica si la ficha está bloqueada (no extraíble).
- **bonusUsada** (boolean): Señala si la bonificación ya fue utilizada.

Descripción de los métodos:

- **Celda(TipoBonificacion bonificacion)**
  Constructor que inicializa la celda con la bonificación indicada; empieza vacía, desbloqueada y con la bonificación disponible.
- **boolean estaOcupada()**
  Devuelve true si la celda contiene una ficha; false si está vacía.
- **TipoBonificacion getBonificacion()**
  Retorna el tipo de bonificación de la celda.
- **boolean isDobleTriplePalabra()**
  true si la bonificación es DOBLE_PALABRA o TRIPLE_PALABRA; false en caso contrario.
- **boolean isDobleTripleLetra()**
  true si la bonificación es DOBLE_LETRA o TRIPLE_LETRA; false en caso contrario.
- **Ficha getFicha()**
  Devuelve la ficha actual, o null si no hay ninguna.
- **boolean colocarFicha(Ficha ficha)**
  Coloca la ficha si la celda está libre y no bloqueada; devuelve true si tuvo éxito.

- **Ficha quitarFicha()**
  Quita y devuelve la ficha si no está bloqueada; o null si no hay ficha o está bloqueada.

- **void bloquearFicha()**
  Bloquea la ficha en la celda y marca la bonificación como usada si aún no lo estaba.

- **boolean estaBloqueada()**
  true si la ficha está bloqueada; false en caso contrario.

- **boolean bonusDisponible()**
  true si la bonificación aún no ha sido usada; false si ya fue consumida.

Dawg

Breve descripción:
Representa un DAWG (Directed Acyclic Word Graph), una estructura para validar y buscar palabras eficientemente. Permite cargar fichas válidas, construir el grafo a partir de un listado de palabras y realizar búsquedas y validaciones por prefijo o palabra completa.

Descripción de los atributos:

- **raiz** (Nodo): Nodo raíz del grafo DAWG.
- **fichasValidas** (Set<String>): Conjunto de tokens (letras o combinaciones) que el DAWG reconoce.
- **longitudMaxFicha** (int): Longitud máxima de token para optimizar la tokenización.
- **registro** (Map<Nodo,Nodo>): Auxiliar para registrar y minimizar nodos durante la construcción.

Descripción de los métodos:

- **Dawg(List<String> lineasArchivoBolsa, List<String> lineasArchivo)**
  Constructor que inicializa el DAWG: carga fichas válidas, establece longitudMaxFicha y construye el grafo con todas las palabras del diccionario.
- **Nodo getRaiz()**
  Retorna el nodo raíz del DAWG.
- **List<String> tokenizarPalabra(String palabra)**
  Tokeniza la palabra en los tokens válidos más largos posibles. Devuelve la lista de tokens o una lista vacía si no es válida.
- **boolean buscarPalabra(String palabra)**
  Comprueba si la palabra existe exactamente en el DAWG; devuelve true si es válida.
- **Nodo buscarUltimoNodo(String palabra)**
  Navega por el grafo según la secuencia de tokens de la palabra y devuelve el último nodo alcanzado, o null si el camino no existe.

Ficha

Breve descripción:
Modela una ficha de Scrabble con una letra (o comodín) y una puntuación asociada.

Descripción de los atributos:

- **letra** (String): Letra o símbolo que identifica la ficha.
- **puntuacion** (int): Valor en puntos que aporta la ficha.

Descripción de los métodos:

- **Ficha(String letra, int puntuacion)**
  Constructor que crea una ficha con la letra y puntuación indicadas.
- **String getLetra()**
  Devuelve la letra de la ficha.
- **int getPuntuacion()**
  Devuelve la puntuación de la ficha.

Jugador

Breve descripción:
Representa a un jugador del juego, identificado por un nombre único.

Descripción de los atributos:

- **nombre** (String): Nombre del jugador.

Descripción de los métodos:

- **Jugador(String nombre)**
  Constructor que crea un jugador con el nombre indicado.
- **String getNombre()**
  Retorna el nombre del jugador.
- **void setNombre(String nombre)**
  Actualiza el nombre del jugador.

Partida

Breve descripción:
Gestiona una partida de Scrabble entre dos jugadores: turnos, manos de fichas, puntuaciones, bolsa y tablero. Proporciona la lógica para repartir fichas, colocar y retirar fichas, calcular puntos, cambiar turnos y manejar el estado de la partida.

Descripción de los atributos:

- **idPartida** (String): Identificador único de la partida.
- **recursoPartida** (String): Recurso asociado (por ejemplo, nombre de fichero).
- **partidaAcabada** (int): Estado de la partida (0 en curso, 1 finalizada).
- **fichasJugador1** (List<Ficha>): Manos de fichas del Jugador 1.
- **fichasJugador2** (List<Ficha>): Manos de fichas del Jugador 2.
- **puntosJugador1** (int): Puntuación acumulada del Jugador 1.
- **puntosJugador2** (int): Puntuación acumulada del Jugador 2.
- **turnoJugador** (boolean): true si es turno del Jugador 1, false si es del Jugador 2.
- **bolsa** (Bolsa): Bolsa de fichas de la partida.
- **tablero** (Tablero): Tablero de juego.
- **contadorTurno** (int): Contador de turnos jugados.
- **coordenadasPalabra** (List<Pair<Integer,Integer>>): Coordenadas de las fichas colocadas en el turno actual.

Descripción de los métodos:

- **Partida(String id, List<String> lineasArchivoBolsa, long seed)**
  Inicializa una nueva partida con reparto de 7 fichas a cada jugador y tablero vacío.
- **Partida(String id, List<String> bolsaTexto, List<String> tableroTexto)**
  Recrea un estado de partida desde texto, cargando bolsa y tablero según las cadenas proporcionadas.

- **void setFichasJugador1(String fichas)**
  Asigna las fichas al Jugador 1 a partir de una cadena de texto.

- **Tablero getTablero()**
  Devuelve el tablero de juego.

- **Bolsa getBolsa()**
  Devuelve la bolsa de fichas.

- **String getRecursoPartida()**
  Retorna el recurso asociado a la partida.

- **void setRecursoPartida(String recurso)**
  Actualiza el recurso de la partida.

- **int getContadorTurno()**
  Obtiene el número de turnos jugados.

- **void aumentarContador()**
  Incrementa el contador de turnos en uno.

- **void setContadorTurno(int contador)**
  Establece el contador de turnos a un valor específico.

- **List<Pair<Integer,Integer>> getCoordenadasPalabras()**
  Devuelve las coordenadas de la palabra en curso.

- **void coordenadasClear()**
  Limpia la lista de coordenadas de la palabra actual.

- **void añadirFicha(String letra, int x, int y)**
  Coloca una ficha o comodín en el tablero y registra su posición.

- **void addPuntos(int pts)**
  Suma puntos al jugador cuyo turno es actual.

- **boolean quitarFichaTablero(int x, int y)**
  Retira una ficha del tablero y la devuelve a la mano del jugador actual.

- **void setFicha(Ficha ficha)**
  Añade una ficha a la mano del jugador actual.

- **int getListSize()**
  Devuelve el número de fichas en mano del jugador actual.

- **boolean recuperarFichas()**
  Rellena manos de ambos jugadores hasta 7 fichas, si la bolsa lo permite.

- **void cambiarTurnoJugador()**
  Alterna el turno entre los dos jugadores.

- **void setTurnoJugador(boolean turno)**
  Fija de quién es el turno (true = Jugador 1; false = Jugador 2).

- **Ficha getFicha(int n)**
  Obtiene la ficha en índice n de la mano del jugador actual.

- **Ficha getFichaString(String s)**
  Busca y devuelve la ficha cuya letra coincide con s en la mano actual.

- **void quitarFicha(String s)**
  Elimina la primera ficha con letra s de la mano actual.

- **boolean isBolsaEmpty()**
  Comprueba si la bolsa está vacía.

- **void setFichas(List<Ficha> fichas)**
  Reemplaza la mano actual con la lista de fichas dada.

- **List<String> obtenerFichas()**
  Devuelve una lista de cadenas "Letra Puntuación" de las fichas en mano.

- **boolean getTurnoJugador()**
  Devuelve el turno actual (true = Jugador 1; false = Jugador 2).

- **void bloquearCeldas()**
  Bloquea en el tablero todas las celdas usadas en el turno actual.

- **void setPuntos(int puntos)**
  Asigna la puntuación del jugador actual a puntos.

- **void setPuntosJugador1(int puntos)**
  Fija directamente la puntuación del Jugador 1.

- **void setPuntosJugador2(int puntos)**
  Fija directamente la puntuación del Jugador 2.

- **int getPuntosJugador1()**
  Devuelve la puntuación del Jugador 1.

- **int getPuntosJugador2()**
  Devuelve la puntuación del Jugador 2.

- **List<Ficha> getFichasJugador()**
  Retorna la lista de objetos Ficha de la mano del jugador actual.

- **void setPartidaAcabada()**
  Marca la partida como finalizada.

- **int getPartidaAcabada()**
  Devuelve el estado de la partida (0 en curso, 1 finalizada).

- **String getIdPartida()**
  Retorna el ID único de la partida.

- **void reset(List<String> fichas)**
  Las fichas que se han seleccionadas se devuelven a la bolsa. Y se mezclan las fichas de la bolsa de nuevo.

Tablero

Breve descripción:
Representa el tablero de Scrabble, una matriz de 15×15 celdas con bonificaciones según las reglas clásicas.

Descripción de los atributos:

- **TAMANO** (int constante): Tamaño fijo del tablero (15).
- **celdas** (Celda[][]): Matriz de objetos Celda de 15 filas y 15 columnas.

Descripción de los métodos:

- **Tablero()**
  Constructor que inicializa la matriz de 15×15 celdas con las bonificaciones correspondientes, dejando el tablero vacío.
- **Celda getCelda(int fila, int columna)**
  Devuelve la celda en la posición indicada, o null si está fuera de los límites.
- **Ficha getFicha(int fila, int columna)**
  Obtiene la ficha de la celda indicada, o null si no hay ficha o la posición es inválida.
- **boolean ponerFicha(Ficha ficha, int fila, int columna)**
  Coloca una ficha en la celda indicada; devuelve true si tuvo éxito.
- **boolean ponerComodin(String letra, int fila, int columna)**
  Coloca un comodín con letra especificada; devuelve true si se colocó.
- **Ficha quitarFicha(int fila, int columna)**
  Quita y devuelve la ficha de la celda indicada, o null si está vacía o inválida.
- **void bloquearCelda(int fila, int columna)**
  Bloquea la ficha en la celda indicada para impedir su retirada.
- **boolean esCentroDelTablero(int fila, int columna)**
  true si la posición es (7,7); false en caso contrario.
- **List<String> toListString()**
  Devuelve una lista de cadenas "Letra Puntuación fila columna" para cada ficha colocada.

- **String getLetraCelda(int fila, int col)**
  Retorna la letra de la ficha en la celda, "#" si es comodín, o null si está vacía o inválida.

- **int getPuntuacionCelda(int fila, int col)**
  Devuelve la puntuación de la ficha en la celda, o 0 si no hay ficha o posición inválida.

- **String getBonusCelda(int fila, int col)**
  Retorna un código de dos letras de la bonificación actual, o "US" si ya fue usada, o espacio si no tiene bonificación.

TipoBonficacion

Breve descripción:
Enum que representa las bonificaciones del tablero de Scrabble, cada una con su multiplicador correspondiente.

Descripción de las constantes:

- **NINGUNA**: Sin bonificación (x1).

- **DOBLE_LETRA**: Doble letra (x2).

- **TRIPLE_LETRA**: Triple letra (x3).

- **DOBLE_PALABRA**: Doble palabra (x2).

- **TRIPLE_PALABRA**: Triple palabra (x3).

Descripción de los métodos:

- **int getMultiplicador()**
  Retorna el multiplicador asociado a la bonificación.

Validador

Breve descripción:
Valida las jugadas según las reglas del Scrabble: alineación, conexión con fichas existentes, paso por el centro en la primera jugada, existencia de palabras (principal y secundarias) en el diccionario DAWG y cálculo de puntuación con bonificaciones.

Descripción de los atributos:

- **tablero** (Tablero): Tablero para comprobar posiciones y bonificaciones.
- **diccionario** (Dawg): Diccionario DAWG para validar palabras.
- **hayBloqueada** (boolean): Indica si la jugada usa fichas previamente bloqueadas.

Descripción de los métodos:

- **Validador()**
  Constructor por defecto.
- **int validarPalabra(List<Pair<Integer,Integer>> coordenadasPalabra, Dawg diccionario, Tablero tablero, int contadorTurno)**
  Valida y puntúa una jugada:
  1. Verifica que se haya colocado al menos una ficha.
  2. En el primer turno, exige paso por la casilla central.
  3. Comprueba que las fichas estén alineadas y sean contiguas.
  4. Asegura conexión con fichas existentes (salvo primer turno).
  5. Valida existencia de las palabras formadas (principal y secundarias) en el DAWG.
  6. Calcula la puntuación total aplicando bonificaciones y reglas especiales.
     Si alguna validación falla, lanza una excepción PalabraInvalidaException; de lo contrario devuelve el puntaje de la jugada.

2.3. Controladores

Controlador de Jugador (CtrlJugador)

Breve descripción:
Controlador de sesión de jugador. Gestiona el jugador actual y un posible segundo jugador en el contexto de una partida de Scrabble.

Descripción de los atributos:

- **jugadorActual** (Jugador): Representa al usuario que ha iniciado sesión.
- **segundoJugador** (Jugador): Representa al rival o segundo participante; puede ser null si no está definido.

Descripción de los métodos:

- **CtrlJugador()**
  Constructor. Inicializa ambos jugadores en null.
- **void setJugadorActual(String nombre)**
  Crea un objeto Jugador con el nombre dado y lo asigna a jugadorActual.
- **void setSegundoJugador(String nombre)**
  Crea un objeto Jugador con el nombre dado y lo asigna a segundoJugador.
- **void resetSegundoJugador()**
  Establece segundoJugador a null.
- **String getJugadorActual()**
  Devuelve el nombre del jugador actual, o null si no hay sesión.
- **String getSegundoJugador()**
  Devuelve el nombre del segundo jugador, o null si no está definido.
- **boolean haySesion()**
  true si jugadorActual no es null; de lo contrario false.
- **void clearSesion()**
  Limpia ambas referencias de jugador, dejando la sesión sin usuarios (jugadorActual = null, segundoJugador = null).

Controlador de Partida (CtrlPartida)

Breve descripción:
Controlador de la lógica para una partida de Scrabble. Gestiona la instancia de Partida activa, la validación de palabras, los turnos de juego, la interacción con la IA (algoritmo) y la finalización de la partida.

Descripción de los atributos:

- **partidaActual** (Partida): Objeto Partida que modela el estado completo del juego.
- **dawg** (Dawg): Estructura de datos para validar palabras y asistir a la IA.
- **validador** (Validador): Componente que verifica la validez de las jugadas según reglas de Scrabble.
- **finTurno** (boolean): Indicador interno de si el turno actual ha concluido.
- **isAlgoritmo** (boolean): true si la partida está en modo IA, false si es juego humano-humano.
- **algoritmo** (Algoritmo): Instancia encargada de generar jugadas automáticas cuando isAlgoritmo es true.
- **jugadorAlgoritmo** (boolean): true si el turno actual corresponde a la IA, false si corresponde a un jugador humano.

Descripción de los métodos:

- **CtrlPartida()**
  Constructor. Deja todos los campos en estado inicial (null o false).
- **void crearPartida(int modo, String id, List<String> lineasArchivo, List<String> lineasArchivoBolsa, long seed, boolean jugadorAlgoritmo)**
  Inicializa una nueva partida: construye el DAWG, crea el objeto Partida (repartiendo fichas según semilla seed), y configura si la IA participa (jugadorAlgoritmo).
- **void setPartida(Partida partida, List<String> diccionario, List<String> bolsa)**
  Carga una partida existente en el controlador, ajustando DAWG y estado interno según diccionario y bolsa.

- **Partida getPartida()**
  Devuelve la instancia de Partida actualmente cargada.

- **List<String> obtenerFichas()**
  Devuelve la lista de fichas (letras) de la mano del jugador cuyo turno es actual, en formato de cadenas.

- **int jugarScrabble(int opcion, String input) throws ComandoInvalidoException, PalabraInvalidaException**
  Procesa un comando de juego (colocar ficha, pasar turno, etc.) según opcion e input; delega en Partida y en Validador. Devuelve el resultado o lanza excepción si el comando o la palabra no son válidos.

- **int getPuntosJugador1()**
  Obtiene la puntuación acumulada del Jugador 1 de la partida actual.

- **int getPuntosJugador2()**
  Obtiene la puntuación acumulada del Jugador 2.

- **Tablero obtenerTablero()**
  Retorna el objeto Tablero asociado a la partida.

- **void setPartidaAcabada()**
  Marca la partida actual como finalizada.

- **String getId()**
  Devuelve el identificador de la partida.

- **void setRecursoPartida(String recurso, List<String> diccionario, List<String> bolsa)**
  Actualiza el recurso (diccionario y bolsa) vinculado a la partida.

- **String getRecursoPartida()**
  Recupera el identificador del recurso actualmente asociado a la partida.

- **void clearPartida()**
  Limpia la referencia a la Partida activa, dejando el controlador listo para una nueva partida.

- **void activarAlgoritmo()**
  Activa el modo de juego con IA: instancia el algoritmo y establece el turno de la IA según configuración.

Controlador de Dominio (CtrlDominio)

Breve descripción:
Controlador principal del dominio del juego Scrabble. Gestiona la lógica de alto nivel: registro e inicio de sesión de jugadores, mantenimiento de sesiones individuales y dobles, gestión de partidas (creación, guardado, carga y abandono), manipulación de recursos (diccionarios y bolsas), delegación al controlador de persistencia y conversión de datos.

Descripción de los atributos:

- **ctrlPersistencia** (CtrlPersistencia): Encargado de las operaciones de lectura y escritura en almacenamiento (jugadores, partidas, recursos, imágenes).

- **ctrlJugador** (CtrlJugador): Controlador de la sesión de jugador actual y segundo jugador.

- **ctrlPartida** (CtrlPartida): Controlador de la lógica de partidas de Scrabble.

- **dc** (DataConverter): Componente interno para transformar datos entre los modelos de dominio (Partido) y los formatos usados por la persistencia (Listas de String).

Descripción de los métodos:

- **CtrlDominio()**
  Constructor. Inicializa los controladores de persistencia, jugador, partida y el conversor de datos.

- **void registrarJugador(String nombre, String password) throws IOException, UsuarioYaRegistradoException**
  Registra un nuevo usuario con nombre y contraseña; lanza excepción si ya existe o falla el acceso a disco.

- **void iniciarSesion(String nombre, String password) throws UsuarioNoEncontradoException, PasswordInvalidaException, InicioSesionIncorrectoException**
  Inicia la sesión del primer jugador validando credenciales. Impide el usuario especial "propAI".

- **void iniciarSesionSegundoJugador(String segundoJugador, String password) throws UsuarioNoEncontradoException, PasswordInvalidaException, InicioSesionIncorrectoException**
  Registra el segundo jugador en la sesión, validando que exista, la contraseña coincida y verifica que el segundo jugador no sea el mismo que el jugador actual

activo en la sesión.

- **void cerrarSesion()**
  Cierra la sesión actual, borrando tanto el jugador principal como el segundo jugador.

- **boolean haySesion()**
  Devuelve true si hay un jugador principal logueado.

- **String getUsuarioActual()**
  Retorna el nombre del jugador principal que tiene la sesión iniciada, o null si no hay sesión.
- **String getSegundoJugador()**
  Obtiene el nombre del segundo jugador, que actúa como oponente del jugador principal durante la partida en curso.

- **int getPuntosActual() throws UsuarioNoEncontradoException**
  Recupera la puntuación máxima registrada del usuario actual en el sistema de persistencia.

- **void cambiarPassword(String antigua, String nueva) throws UsuarioNoEncontradoException**
  Modifica la contraseña del jugador actual si la antigua coincide.

- **void cambiarNombre(String nuevoNombre, String password) throws UsuarioNoEncontradoException, IOException, UsuarioYaRegistradoException, PasswordInvalidaException**
  Cambia el nombre de usuario de la sesión actual, actualizando además el registro en persistencia; valida contraseña.

- **void eliminarUsuario(String password) throws UsuarioNoEncontradoException, IOException**
  Borra el usuario activo, previa validación de contraseña.

- **List<Map.Entry<String, Integer>> obtenerRanking() throws RankingVacioException, IOException**
  Devuelve la lista ordenada de jugadores y sus puntuaciones máximas.

- **int obtenerPosicion(String nombre) throws UsuarioNoEncontradoException, IOException**
  Devuelve la posición en el ranking de un jugador dado.

- **void salirPartida() throws UsuarioNoEncontradoException**
  Hace que el jugador actual abandone la partida actual.

- **void guardarPartida(String id) throws UsuarioNoEncontradoException**
  Persiste el estado de la partida actual bajo el identificador id.

- **List<String> obtenerNombresPartidasGuardadas()**
  Lista los identificadores de todas las partidas no acabadas del jugador actual.

- **void eliminarPartidaGuardada(String id) throws PartidaNoEncontradaException**
  Elimina el recurso de guardado de la partida cuyo id se proporciona.

- **int jugarScrabble(int modo, String id) throws PuntuacionInvalidaException, ComandoInvalidoException, PalabraInvalidaException, UsuarioNoEncontradoException, PartidaYaExistenteException, UltimaPartidaNoExistenteException**
  Inicia o retoma una partida de Scrabble en el modo indicado (modo) y con identificador id. Devuelve el resultado (código de finalización o puntuación).

- **List<String> obtenerFichas()**
  Obtiene la representación en texto de las fichas en mano del jugador cuyo turno es actual en la partida.

- **int getPuntosJugador1()**
  Recupera la puntuación acumulada del Jugador 1 en la partida en curso.

- **int getPuntosJugador2()**
  Recupera la puntuación acumulada del Jugador 2.

- **int getTableroDimension()**
  Devuelve la dimensión (ancho/alto) del tablero de Scrabble (constante).

- **String getLetraCelda(int fila, int col)**
  Retorna la letra de la ficha en la celda indicada del tablero, o null si está vacía o es inválida.

- **int getPuntuacionCelda(int fila, int col)**
  Devuelve la puntuación de la ficha en la celda especificada, o 0 si no hay ficha.

- **String getBonusCelda(int fila, int col)**
  Obtiene el código de bonificación de la celda (por ejemplo, "DL", "TP", "US", etc.).

- **List<String> obtenerRecursos()**
  Lista los identificadores de todos los recursos (pares diccionario/bolsa) disponibles.

- **void crearRecurso(String id, List<String> diccionario, List<String> bolsa) throws IOException, RecursoExistenteException,**

FormatoDiccionarioInvalidoException, FormatoBolsaInvalidoException
Crea un nuevo recurso con identificador id, validando formatos de diccionario y bolsa.

- **void modificarRecurso(String id, List<String> diccionario, List<String> bolsa) throws IOException, RecursoNoExistenteException, BolsaNoEncontradaException, DiccionarioNoEncontradoException, FormatoDiccionarioInvalidoException, FormatoBolsaInvalidoException**
  Actualiza el diccionario y/o bolsa de un recurso existente.

- **void eliminarRecurso(String id) throws IOException**
  Elimina un recurso de diccionario y bolsa.

- **List<String> obtenerDiccionario(String id) throws DiccionarioNoEncontradoException, IOException**
  Recupera la lista de palabras del diccionario asociado a id.

- **List<String> obtenerBolsa(String id) throws BolsaNoEncontradaException, IOException**
  Recupera la configuración de la bolsa asociada a id.

- **void validarFormatoDiccionario(List<String> diccionario) throws FormatoDiccionarioInvalidoException**
  Comprueba que cada línea del diccionario siga el formato "PALABRA" y orden alfabético.

- **void saveProfileImage(BufferedImage image)**
  Almacena la imagen de perfil del jugador actual.

- **BufferedImage getProfileImage(String username)**
  Recupera la imagen de perfil asociada a username.

- **void deleteProfileImage(String username)**
  Elimina la imagen de perfil del usuario indicado.

2.4. Clases auxiliares

Nodo

Representa un nodo en un DAWG (Directed Acyclic Word Graph). Cada nodo almacena una letra o ficha identificadora (o null si es la raíz), un mapa de hijos que apuntan a nodos descendientes, y un indicador de si marca el final de una palabra válida.

Descripción de los atributos:

- **letra** (String): Letra o ficha que identifica este nodo; es null en el nodo raíz.
- **hijos** (Map<String, Nodo>): Mapa que asocia cada cadena (ficha) con su nodo hijo correspondiente.
- **palabraValidaHastaAqui** (boolean): Indica si este nodo corresponde al fin de una palabra válida en el DAWG.

Descripción de los métodos:

- **Nodo(String letra)**
  Constructor que inicializa este nodo con la letra especificada; los hijos comienzan vacíos y palabraValidaHastaAqui es false.
- **Nodo()**
  Constructor de nodo raíz. Invoca al anterior con letra = null.
- **boolean esValida()**
  Devuelve true si palabraValidaHastaAqui es true (marca final de palabra); false en caso contrario.
- **String getLetra()**
  Retorna la letra asociada al nodo, o null si es la raíz.
- **Map<String, Nodo> getHijos()**
  Devuelve el mapa de hijos de este nodo.
- **boolean equals(Object obj)**
  Compara este nodo con otro: son iguales si tienen la misma letra (o ambas null), el mismo estado de fin de palabra y el mismo conjunto de hijos (claves y referencias).
- **int hashCode()**
  Genera un código hash consistente con equals(), basado en la letra, el estado de fin de palabra y las entradas del mapa de hijos.

Pair

Breve descripción:
Modelo genérico de un par de valores, first y second. Útil para representar tuplas o pares heterogéneos de objetos.

Descripción de los atributos:

- **first** (F): Primer elemento del par.
- **second** (S): Segundo elemento del par.

Descripción de los métodos:

- **Pair(F first, S second)**
  Constructor que inicializa el par con los valores proporcionados.
- **F getFirst()**
  Retorna el primer elemento del par.
- **S getSecond()**
  Retorna el segundo elemento del par.
- **String toString()**
  Devuelve la representación en texto del par en formato "(first, second)".
- **boolean equals(Object obj)**
  Compara este par con otro: son iguales si ambos elementos (first y second) son iguales según equals().
- **int hashCode()**
  Genera un código hash consistente con equals(), basado en ambos elementos.

DataConverter

Breve descripción:
Clase encargada de convertir una partida a una lista de String (serialización) y de reconstruir (deserializar) un objeto Partida a partir de esa lista. Facilita el almacenamiento y recuperación del estado completo de la partida en un formato sencillo de cadenas.

Descripción de los atributos:
Esta clase no define atributos de instancia.

Descripción de los métodos:

- **List<String> partidaToStringList(Partida partida, String jugadorActual, String segundoJugador, String nombrePartida)**
  Convierte el estado de una partida en curso en una lista de cadenas. La lista contiene, en orden fijo:

  1. Estado de la partida (0 en curso, 1 finalizada).
  2. Nombre de la partida.
  3. Contador de turnos jugados.
  4. Turno actual ("1" si es turno del jugador 1, "0" si es turno del jugador 2).
  5. Nombre del jugador 1.
  6. Nombre del jugador 2.
  7. Puntuación acumulada del jugador 1.
  8. Puntuación acumulada del jugador 2.
  9. Fichas en mano del jugador cuyo turno es actual (cada ficha como "Letra Cantidad Puntuacion").
  10. Fichas en mano del otro jugador (tras cambiar temporalmente el turno internamente).
  11. Bolsa de fichas restante (primer elemento: número de entradas, seguido de cada "Letra Cantidad Puntuacion").
  12. Estado del tablero (primer elemento: número de celdas con ficha, seguido de cada entrada "Letra Puntuacion fila columna" o "null").

13. Identificador del recurso usado (bolsa + diccionario).

- **Partida stringListToPartida(List<String> strings)**
  Reconstruye un objeto Partida a partir de la lista de cadenas generada por partidaToStringList. El método:

  1. Extrae nombres de jugador y nombre de partida.
  2. Obtiene el estado de turno y contador de turnos.
  3. Recupera las puntuaciones de ambos jugadores.
  4. Reconstruye la lista de fichas de cada jugador leyendo las cadenas correspondientes y creando objetos Ficha.
  5. Carga la configuración de la bolsa y del tablero a partir de sus porciones en la lista (respetando tamaños y secuencias).
  6. Crea la instancia de Partida con constructor adecuado, ajusta turnos, fichas, puntuaciones y recurso.
  7. Devuelve la partida completamente restaurada, lista para reanudar el juego.

2.5. Excepciones

A continuación se recoge el catálogo completo de excepciones utilizadas por nuestro sistema para gestionar los casos anómalos que pueden producirse durante la ejecución.

Cada una modela una situación de error muy específica, lo que facilita el diagnóstico y la corrección de fallos tanto en tiempo de desarrollo como en producción.

Algunas, por ejemplo *RankingVacioException*, no muestran ningún aviso visual al jugador: se lanzan únicamente para que el sistema detecte estados inconsistentes y tome las medidas oportunas. Otras sí se traducen en mensajes de interfaz para guiar al usuario.

BolsaNoEncontradaException

Indica que se ha intentado acceder a una bolsa de fichas que no está registrada en el sistema, por ejemplo al iniciar una partida con un identificador de bolsa inexistente.

BolsaYaExistenteException

Se produce cuando se intenta registrar una nueva bolsa de fichas con un identificador que ya pertenece a otra bolsa existente, evitando la duplicación de recursos.

ComandoInvalidoException

Señala que el comando introducido no forma parte del conjunto de instrucciones admitidas o presenta una sintaxis incorrecta. Se lanza para interrumpir la operación y notificar el error.

DiccionarioNoEncontradoException

Indica que se ha solicitado un diccionario que no existe en el sistema, ya sea al configurar una partida o al intentar visualizarlo desde las opciones de administración.

DiccionarioYaExistenteException

Se genera cuando se intenta crear o importar un diccionario con un identificador que ya está en uso, impidiendo la sobrescritura involuntaria.

FormatoBolsaInvalidoException

Informa de que los datos proporcionados para definir el contenido de una bolsa de fichas no cumplen el formato exigido (por ejemplo, caracteres no válidos o distribución incorrecta).

FormatoDiccionarioInvalidoException

Se lanza cuando el archivo o la cadena utilizada para crear un diccionario no respeta el formato requerido (cabeceras, codificación o separadores erróneos).

InicioSesionIncorrectoException

Señala que la combinación de usuario y contraseña introducida al iniciar sesión no es válida. Salta en el caso que el segundo jugador inicia sesión con las mismas credenciales del jugador/usuario principal en el modo de dos juagdores.

NoHayPartidaGuardadaException

Indica que el usuario ha solicitado continuar una partida, pero no existe ninguna partida guardada en su perfil.

PalabraInvalidaException

Se lanza cuando la palabra introducida no es válida según el diccionario activo, contiene caracteres ilegales o incumple las normas de longitud o acentuación.

PartidaNoEncontradaException

Señala que se ha intentado cargar, reanudar o eliminar una partida cuyo identificador no coincide con ninguna partida guardada.

PartidaYaExistenteException

Se produce al intentar guardar una nueva partida con un identificador que ya está en uso, evitando sobrescribir datos sin confirmación explícita.

PasswordInvalidaException

Informa de que la contraseña suministrada no coincide con la almacenada durante operaciones sensibles como cambio de clave o eliminación de la cuenta.

PuntuacionInvalidaException

Señala que se ha intentado registrar una puntuación que no respeta las reglas del juego (por ejemplo, negativa o superior al máximo posible).

RankingVacioException

Indica que se ha solicitado visualizar el ranking de puntuaciones pero aún no hay registros disponibles.

RecursoExistenteException

Se lanza al intentar crear un recurso genérico (fichas, diccionario, partida, etc.) cuyo identificador ya existe y no encaja en una categoría más específica.

RecursoNoExistenteException

Indica que se ha solicitado un recurso genérico que no está registrado en el sistema y que no corresponde a una sub-clase concreta de excepción.

UltimaPartidaNoExistenteException

Informa de que se ha pedido reanudar la última partida disputada, pero el usuario aún no ha jugado ninguna o no se ha guardado la más reciente.

UsuarioNoEncontradoException

Indica que se ha solicitado un usuario que no existe en el sistema. Se lanza, por ejemplo, al iniciar sesión con un nombre no registrado o al intentar operar sobre una cuenta inexistente.

UsuarioYaRegistradoException

Señala que se está intentando crear un perfil con un nombre que ya existe. Aparece durante el proceso de registro y detiene la operación hasta que se elija otro identificador.

3. Capa de Persistencia

3.1. Diagrama de gestión de disco

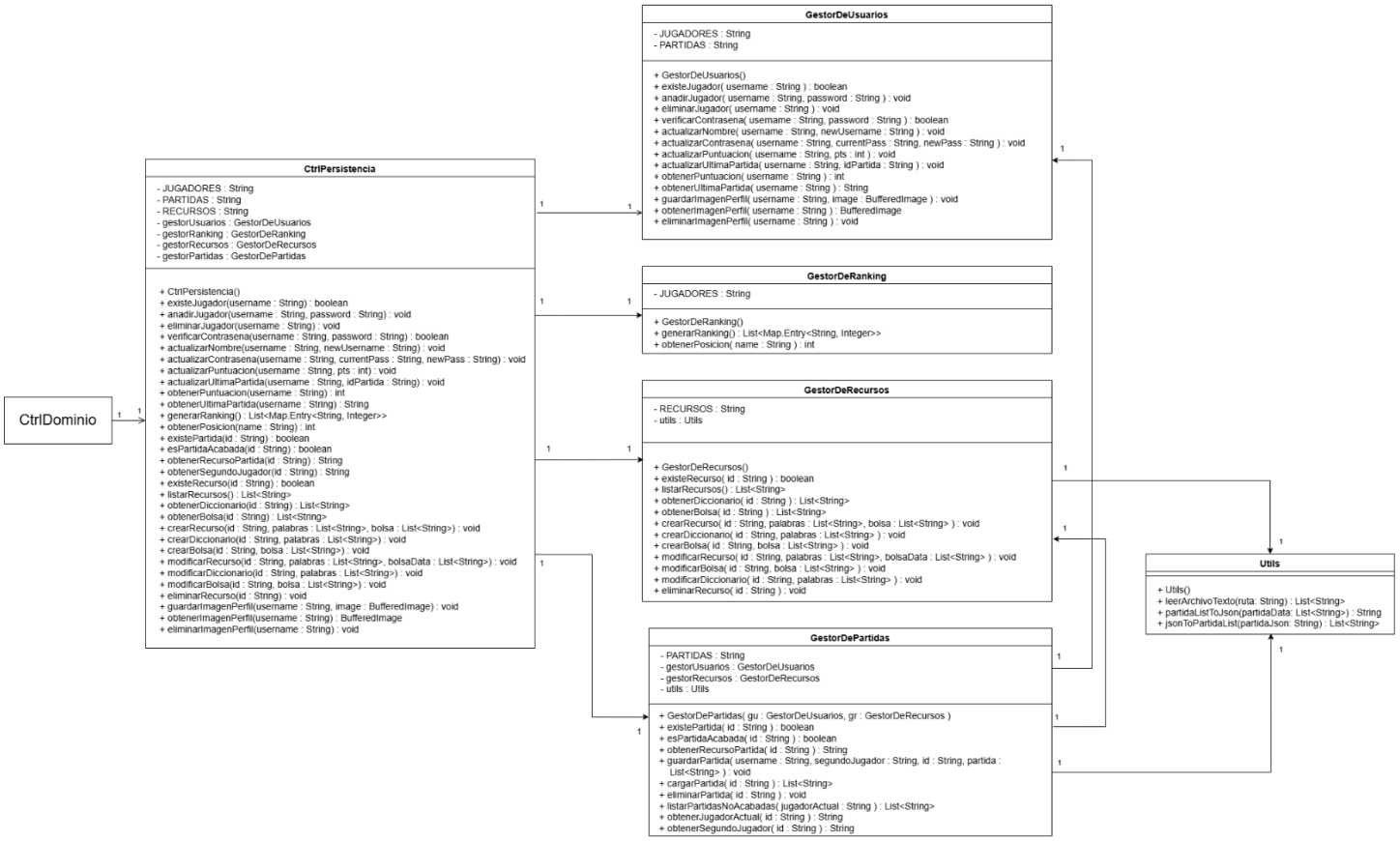

fig. 2 Diagrama de la Capa de Persistencia, del sistema implementado (Se encuentra en la carpeta DOCS).

3.2. Controlador

Controlador de Persistencia (CtrlPersistencia)

Nombre de la clase: CtrlPersistencia
Breve descripción:
Controlador de la capa de persistencia. Gestiona usuarios (creación, autenticación, eliminación), ranking de jugadores, almacenamiento y recuperación de partidas, recursos de diccionario y bolsa, y manejo de imágenes de perfil. Utiliza la librería **org.json** para serializar y deserializar datos en formato JSON.

3.3. Gestores de Disco

Gestor de Usuarios

Nombre de la clase: GestorDeUsuarios
Breve descripción:
Controlador de persistencia especializado en la gestión de la información de los jugadores. Se encarga de crear y eliminar cuentas de usuario, verificar credenciales, actualizar datos de perfil (nombre de usuario, contraseña, puntuación y última partida jugada) y manejar la imagen de perfil asociada a cada jugador.

Gestor de Ranking

Nombre de la clase: GestorDeRanking
Breve descripción:
Controlador encargado de construir y consultar el ranking de jugadores según sus puntuaciones almacenadas en el sistema de archivos. Lee los directorios de usuarios, extrae sus puntuaciones de los ficheros JSON y entrega tanto la lista ordenada de pares (usuario, puntos) como la posición individual de un jugador en ese ranking.

Gestor de Partidas

Nombre de la clase: GestorDePartidas
Breve descripción:
Controlador de persistencia para las partidas de Scrabble. Gestiona la existencia, almacenamiento, recuperación y eliminación de archivos JSON que representan el estado de cada partida, así como la consulta de partidas en curso y la obtención de los jugadores implicados.

Gestor de Recursos

Nombre de la clase: GestorDeRecursos
Breve descripción:
Controlador responsable de la gestión en persistencia de los "recursos" de juego (pares diccionario+bolsa). Se encarga de verificar la existencia, listar todos los recursos disponibles, cargar ("obtener") el contenido de diccionarios y bolsas, y crear, modificar o eliminar completamente un recurso en el directorio de recursos.

3.4. Utilidades

Utilidades para los Gestores (Utils)

Nombre de la clase: Utils,java
Breve descripción:
Clase de utilidades para la lectura de archivos de texto y la conversión entre la representación en lista de cadenas de datos de una partida y su formato JSON, usando la librería org.json.

4. Capa de Presentación

La clase **CtrlPresentacion** actúa como el único punto de entrada y orquestador de toda la capa de presentación: Se crea cada vista (login, menú principal, lateral, partidas, juego, ranking, cuenta, recursos, manual, etc.) y las añade a un contenedor central (**VistaPrincipal**).

Cuando el usuario interactúa con un botón o menú, una función privada invoca al **CtrlDominio** para ejecutar la lógica de negocio (iniciar sesión, crear/cargar partida, jugar turnos, obtener datos de ranking o perfil) y se actualiza la vista correspondiente, configurando componentes, rellenando tablas o cambiando de vista en **VistaPrincipal**.

Todos los métodos que definen la navegación y la construcción de vistas son **privados** porque forman parte de la implementación interna del controlador de presentación: no deben usarse directamente desde otras clases.

Este encapsulamiento garantiza que todo el flujo de pantallas quede centralizado y controlado por **CtrlPresentacion**, evitando llamadas dispersas y preservando la clara separación entre la capa de presentación y la capa de dominio (**CtrlDominio**).

Además, al instanciar las vistas solo cuando se necesitan, se optimiza el uso de recursos y se simplifica el mantenimiento del ciclo de vida de cada componente visual.

4.1. Diagrama de Capa de Presentación

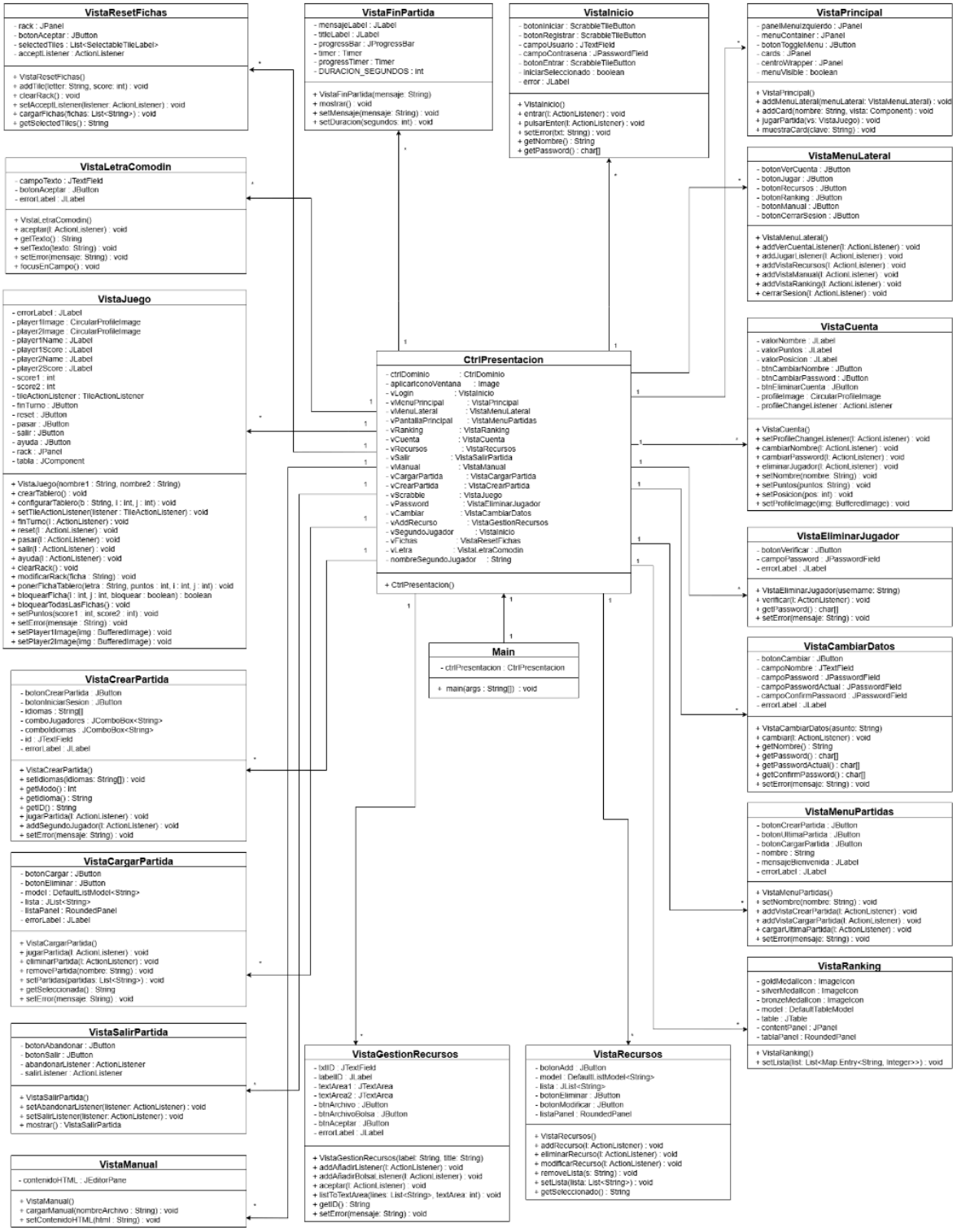

fig. 3 Diagrama de la Capa de Presentación, del sistema implementado (Se encuentra en la carpeta DOCS).

4.2. Controlador de Presentación

Clase: CtrlPresentacion.java

Controlador principal de la capa de presentación. Se encarga de gestionar la lógica de navegación entre vistas, inicialización de ventanas, y la comunicación con el controlador de dominio. Es la clase central que orquesta qué vista se muestra en cada momento según la interacción del usuario.

Características:

- Controlar el flujo de la aplicación desde la presentación.
- Mostrar las distintas vistas del programa (VistaLogin, VistaCuenta, VistaRecursos, etc.).
- Delegar llamadas al **controlador de dominio** (CtrlDominio) cuando se necesita acceder a la lógica del juego o los datos del sistema.
- Administrar el estado del usuario (quién está logueado, partida en curso, etc.).
- Cargar y actualizar vistas según acciones del usuario (crear partida, cambiar nombre, ver ranking, etc.).

4.3. Vistas

VistaCambiarDatos
Vista pop-up para que el jugador actualice nombre de usuario y/o contraseña (esta vista es genérica y se modifica según el caso de uso: Cambiar nombre o cambiar contraseña), se notifica al usuario si la contraseña es errónea o si el nombre ya existe.

VistaCargarPartida
Vista que lista partidas guardadas y permite reanudarlas o eliminarlas, mostrando los identificadores de las partidas en orden lexicográfico.

VistaCrearPartida
Vista para crear una nueva partida donde se elige el identificador, el modo y el recurso.

VistaCuenta
Vista informativa con los datos básicos del jugador (nombre, foto de perfil y puntuación máxima) y opciones para gestionar la cuenta (cambiar nombre, cambiar contraseña y eliminar jugador).

VistaEliminarJugador
Vista pop-up de doble confirmación que solicita la contraseña antes de borrar la cuenta y todos sus datos.

VistaFinPartida
Vista pop-up final que declara al ganador al terminar o abandonar una partida.

VistaGestionRecursos
Vista administrativa para crear o editar bolsas y diccionarios instalados en el sistema.

VistaInicio
Vista de bienvenida con formulario para iniciar sesión o registrar un nuevo usuario. También se usa para iniciar sesión o registrar un segundo jugador al jugar en modo duo (2 jugadores)

VistaJuego
Vista principal de la partida: información de los jugadores durante la partida (nombres, puntuaciones, avatares), tablero, atril (rack: fichas en mano del jugador), panel de puntos, y controles de turno integrados en una única vista.

VistaLetraComodin
Vista pop-up donde el jugador selecciona la letra que representará un comodín antes de colocarlo.

VistaManual
Vista del manual del juego (con un visor de HTML) con visor HTML, los manuales estarán en la carpeta FONTS/src/main/Recursos/Manuales/.

VistaMenuLateral
Panel lateral que permite navegar entre vistas, y orfrece las opciones de Ver Cuenta, Jugar, Ver Ranking, Recursos, Manual y Salir. Se puede abrir y cerrar.

VistaMenuPartidas
Vista que reúne las opciones de crear una partida, cargar última partida o cargar otras partidas.

VistaPrincipal
Marco global de la aplicación que intercambia las distintas vistas dentro de un contenedor central, mediante el menú lateral.

VistaRanking
Vista que muestra el ranking global de jugadores según sus puntuaciones máximas, en orden descendente. Se muestra la posición, el nombre y la puntuación.

VistaRecursos
Vista que lista todos los recursos disponibles bolsas y diccionarios instalados, se ofrecen las opciones de añadir un nuevo recurso, modificar o eliminar un recurso (3 botones).

VistaResetFichas
Vista pop-up para seleccionar las fichas del atril y cambiarlas por nuevas sacadas de la bolsa.

VistaSalirPartida
Vista que ofrece al jugador dos opciones: finalizar la partida de forma definitiva o salir guardando automáticamente el progreso como partida pendiente.

5. Estructura de datos y algoritmos utilizados

5.1. Estructuras de datos

Pair<L,R>

Contenedor genérico que solo agrupa dos valores (**Left** y **Right**) sin más semántica para cuando se necesita devolver o pasar un par de objetos sin crear una clase más.

List<Object>

La hemos usado en la clase Bolsa, el campo **List<Ficha> conjuntoDeFichas** mantiene la reserva activa de letras: una colección ordenada y mutable que admite duplicados

DAWG (Grafo acíclico dirigido de palabras)

La clase **DAWG** representa un grafo acíclico dirigido construido a partir de un conjunto de palabras válidas. Internamente, utiliza un nodo raíz, una colección de fichas válidas (HashSet), un registro de subgrafos (HashMap) y nodos conectados recursivamente (Map<String, Nodo>). Esta estructura permite validar y buscar palabras de forma eficiente, aprovechando la compresión estructural del grafo al compartir sufijos comunes entre palabras.

Map<String, Nodo> (hijos de un Nodo)

Dentro de la clase **Nodo**, que forma parte del DAWG (Directed Acyclic Word Graph), se utiliza un **Map<String, Nodo>** para almacenar los hijos del nodo actual. Esta estructura permite representar conexiones desde un nodo hacia sus siguientes posibles letras. La elección de un mapa proporciona una forma eficiente de navegación y construcción del grafo al permitir acceso rápido por clave (letra).

HashSet<String> (fichasValidas)

En la clase del Dawg, se ha utilizado un **HashSet<String>** para almacenar las fichas válidas según los datos de configuración de la bolsa. Esta estructura permite validar de forma eficiente (O(1) promedio) si una ficha introducida por el usuario o generada por el algoritmo es válida. Es especialmente útil durante la construcción del DAWG y en el proceso de búsqueda y validación de palabras.

HashMap<String, Nodo> (Registro de los subgrafos)

La clase Dawg emplea un **HashMap<String, Nodo>** llamado registro, que actúa como una tabla de memorización durante la construcción del grafo. Cada entrada en este mapa permite detectar subgrafos ya existentes que pueden ser reutilizados, lo que evita

duplicación de nodos y asegura que el grafo resultante sea compacto, eficiente y sin redundancias.

Nodo (Estructura recursiva para el DAWG)

La clase **Nodo** es una estructura de datos recursiva diseñada para representar los vértices de un DAWG. Cada nodo puede tener múltiples hijos (otras instancias de **Nodo**) y un indicador booleano que señala si ese nodo corresponde al final de una palabra válida.

List<Map.Entry<String, Integer>> (Ranking)

Se ha utilizado una **List<Map.Entry<String, Integer>>** para representar el ranking de jugadores, donde cada entrada contiene un nombre de usuario (clave String) y su puntuación máxima (valor Integer). Esta estructura permite ordenar fácilmente el ranking según las puntuaciones, ya que las listas son directamente comparables y manipulables. También permite mostrar el ranking de forma descendente y realizar búsquedas o filtrados según los criterios necesarios para la visualización. La lista se construye a partir del **Map<String, Integer>** que almacena el ranking en bruto, facilitando así su uso en la interfaz gráfica y en la persistencia.

5.2. Algoritmos

5.2.1. Grafo Dirigido Acíclico de Palabras

1. Definición del DAWG

Un **DAWG** (o en castellano Grafo Dirigido Acíclico de Palabras) es una estructura de datos para almacenar eficientemente un gran diccionario de palabras, compartiendo sufijos comunes, el cual usaremos como diccionario en nuestro Scrabble. S

Su principal característica es que minimiza la redundancia al fusionar subárboles idénticos, reduciendo drásticamente el número de nodos y, por tanto espacio en la memoria y garantizando una rapidez de búsqueda.

2. Componentes básicos

- **Nodo**: representa un estado en el grafo; contiene un mapa de hijos (**hijos**) indexado por fichas (de tipo String) y un booleano **palabraValidaHastaAqui**.

- **Fichas válidas**: conjunto de cadenas (por ejemplo, “A”, “CH”, “LL”) que el DAWG acepta como unidades de tokenización.

- **Registro de minimización**: un **HashMap<Nodo, Nodo>** que mantiene nodos ya procesados y permite fusionar rápidamente nuevos nodos equivalentes.

3. Flujo principal del DAWG

1. **Carga de fichas (cargarFichasValidas)**

    - Lee un listado y extrae la primera columna como ficha y guarda en un **HashSet**, el fichero tiene que seguir las indicaciones mencionadas por el profesorado.

    - Calcula la longitud máxima **L** que pueden tener las fichas.

2. **Tokenización de una palabra**

    - Recorre la palabra (longitud n) de izquierda a derecha.

    - Para cada posición, prueba subcadenas decrecientes de tamaño hasta L (lo máximo que puede tener la ficha, parámetro calculado previamente en cargarFichasVálidas), buscando la más larga que exista en el conjunto de fichas.

    - Si no encuentra ninguna, la tokenización falla; en caso contrario, avanza el índice y repite.

3. **Construcción incremental minimizada**
    - Asume que la lista de palabras (W palabras, una por cada línea) ya viene ordenada lexicográficamente.
    - Para cada nueva palabra:
        1. **Tokenizar** (coste ~O(m·L), m es la longitud de la palabra).
        2. **Detectar prefijo común** con la palabra anterior (O(min(m, m□))). por lo que como máximo se hacen comparaciones igual al tamaño de la palabra más corta
        3. **Minimizar** todos los nodos correspondientes al sufijo que ya no comparten prefijo: por cada nodo sobrante, comprueba en el registro; si existe un equivalente, "apunta" al nodo compartido, si no, lo añade al registro. Esto toma tiempo proporcional al número de nodos a procesar, O(s).
        4. **Insertar** los tokens restantes como nuevos nodos en el grafo (O(t), con t = tokens nuevos).
    - Tras procesar todas las palabras, minimiza el camino de la última palabra de la misma forma.

Ejemplo simple para explicar el proceso + coste, "CASA" → "CASO"

Prefijo común

- Anterior = **C A S A**
- Nueva = **C A S O**
- Prefijo = **C A S** (3 letras/tokens/nodos comunes)

Minimizar sufijo anterior

- Sufijo de "CASA" más allá del prefijo: **A**
- Compruebo en el registro si ya existe un nodo equivalente a ese "A" terminal.
- Si existe, lo apunto; si no, lo guardo.
- Coste**:** O(s)=O(1) (s = 1 nodo)

Insertar sufijo nuevo

- Sufijo de "CASO" más allá del prefijo: **O**
- Creo un nuevo nodo "O" marcado como final y lo enlazo.
- Coste: O(t)=O(1) (t = 1 nodo)

```
Unset
(raíz) --C--> [C] --A--> [A] --S--> [S] --A* (CASA)
                                  \
                                   --> O* (CASO)
```

4. **Búsquedas**
    - **Palabra completa**: tokeniza (O(m·L)) y recorre los hijos; al final comprueba palabraValidaHastaAqui. (m es la longitud de la palabra).
    - **Prefijo**: igual que palabra, pero sólo importa llegar a un nodo válido.
    - **Último nodo**: devuelve el nodo alcanzado tras tokenizar; null si algún paso falla.

4. Análisis de complejidad

- **Parámetros clave**
    - W: número de palabras a cargar.
    - m: longitud media de cada palabra (en caracteres).
    - L: longitud máxima de ficha (token).

- **cargarFichasValidas**
    - Recorre todas las líneas (digamos M): **O(M)**.

- **tokenizarPalabra**
    - En cada uno de sus m caracteres prueba hasta L subcadenas y busca en un HashSet (O(1) medio): **O(m·L)**.

- **construirDesdeArchivo**
    - Para cada palabra (total W):

        1. Tokenizar: **O(m·L)**
        2. Prefijo común: **O(m)**
        3. Minimización de sufijos: **O(m)**
        4. Inserción de nuevos nodos : **O(m)**

    - Como m y L suelen acotarse (longitud razonable de palabra y ficha), el coste real crece **aproximadamente lineal** en el número de palabras: **O(W·m·L)** (o, para m,L constantes, entonces **O(W)**).

- **buscarPalabra / buscarPrefijo / buscarUltimoNodo**
    - Cada uno tokeniza (O(m·L)) y desciende por a lo sumo m hijos (O(1) medio por acceso): **O(m·L)**.

5.2.2. Algoritmo de resolución del Scrabble (Jugador máquina)

1. Explicación

El algoritmo tiene como objetivo identificar la mejor palabra disponible dado un tablero con unas fichas colocadas y una lista de máximo 7 fichas del jugador, que proporcione la mayor cantidad de puntos de una manera eficiente. Para ello, combina el valor intrínseco de cada ficha con las bonificaciones del tablero (letras dobles/triples, palabras dobles/triples y el bono de 50 puntos por usar todas las fichas).

Está inspirado en la estrategia de Appel & Jacobson (1988), que demuestra cómo, gracias a la simetría del tablero, basta plantear la búsqueda para una sola dirección (por ejemplo, horizontal) y, con un simple cambio de orientación, repetirla en vertical. De este modo, el problema se reduce a una sola dimensión: generamos movimientos sobre filas (o columnas, trasponiendo el tablero) y aprovechamos esa abstracción a lo largo de todo el proceso.

La estrategia del algoritmo consiste en partir de las llamadas **celdas ancla** ("anchors"), que son las casillas vacías adyacentes a fichas ya existentes en el tablero. A partir de cada ancla se exploran primero los posibles prefijos situados antes de ella usando únicamente fichas de la mano y después se extienden esos prefijos hacia la derecha, letra a letra, gracias a la estructura de datos DAWG que actúa como diccionario en memoria. En cada paso de la extensión se valida la formación de palabras perpendiculares (los "cross-checks"), y siempre que se alcanza un nodo terminal del DAWG en una casilla vacía se calcula la puntuación completa de la palabra resultante, actualizando la mejor jugada si supera el récord previo.

2. Componentes básico

- **DAWG.** Un grafo acíclico y mínimo que alberga todo el diccionario en memoria. Cada nodo representa un prefijo válido y cada arista, una letra o diptongo; los nodos terminales señalan palabras completas.
- **Tablero.** El tablero es una matriz de 15 × 15 celdas que pueden contener fichas y poseen bonificadores individuales de letra o palabra. El algoritmo interactúa con él para consultar si cada casilla está vacía u ocupada, leer las fichas ya colocadas adyacentes, aplicar los multiplicadores correspondientes y calcular también los puntos de las palabras perpendiculares que se formen al añadir nuevas letras.
- **Lista de fichas del jugador.** La mano del jugador consiste en un máximo de siete fichas, cada una identificada por su letra (o blank) y un valor en puntos. Durante la exploración, las fichas se retiran y reponen de esta lista para probar todas las combinaciones posibles que encajen en el DAWG.

3. Flujo principal del Algoritmo.

El núcleo de la búsqueda se organiza en cinco fases claramente diferenciadas. A continuación se describen en detalle cada una de ellas, numeradas como 3.1 a 3.5,

aunque su ejecución se encadena de forma continua hasta hallar la jugada de mayor puntuación.

3.1. **Búsqueda de anclas.** El algoritmo comienza con un recorrido completo del tablero para identificar las celdas ancla, es decir, aquellas casillas vacías que poseen al menos un vecino ocupado (ya sea a la izquierda, derecha, arriba o abajo). Todas las posiciones que cumplen este criterio se almacenan en una lista de anclas; este conjunto de puntos de partida asegura que cada palabra generada se conectará con fichas ya presentes en el tablero, evitando la exploración de jugadas aisladas que no aportan valor. En caso de no encontrar ninguna, significa que el algoritmo está en el turno inicial, cuando el tablero está vacío, en ese caso, usará como ancla el centro del tablero.

3.2. **Construcción inicial de prefijos.** Para cada ancla de la lista, se determina el prefijo fijo situado inmediatamente antes de ella en la dirección de juego (horizontal o, más adelante, vertical). Si existen fichas adyacentes en esa dirección, se extraen de izquierda a derecha hasta llegar a una casilla vacía o al límite del tablero, formando un fragmento que queda fuera del control de la mano. En ausencia de letras previas, el algoritmo calcula cuántas casillas vacías consecutivas hay antes del ancla para establecer un "límite" que indicará cuántas fichas de la mano se podrán usar al crear prefijos.

3.3. **Generación de prefijos en la izquierda/arriba (Before Part).** Con el espacio libre delimitado, arranca la función **beforePart**, que explora recursivamente todas las combinaciones posibles de fichas de la mano para formar prefijos a la izquierda del ancla. Partiendo del nodo raíz del DAWG y con un contador que representa las casillas disponibles, el algoritmo prueba una a una las letras que aún quedan, avanza por la arista correspondiente en el grafo y reduce el límite en uno, retirando temporalmente la ficha. Cada vez que se coloca una nueva letra, se repite el proceso hasta agotar los espacios o las fichas, y en cada paso se invoca la fase de extensión para prolongar ese prefijo hacia la derecha.

3.4. **Extensión hacia la derecha/abajo y validación (Extend After).** La función extendAfter recibe un prefijo ya formado, su nodo asociado en el DAWG y la posición de la casilla vacía situada tras la última letra. A partir de ahí, recorre cada arista hija del nodo actual, comprueba si la letra correspondiente está disponible en la mano y, antes de colocarla, ejecuta un chequeo perpendicular ("cross-check") para asegurar que la

palabra vertical que se formaría es válida. Si el cruce resulta aceptable, avanza al siguiente nodo del DAWG y casilla del tablero, acumula provisionalmente la puntuación de esa letra (incluyendo bonificadores de letra y palabra) y retira la ficha de la mano. Cuando alcanza un nodo terminal en una casilla vacía, significa que ha construido una palabra completa y válida, momento en el cual se pasa al cálculo definitivo de puntos.

3.5. **Cálculo de puntuación y selección final**. Al completarse una palabra válida, el algoritmo recorre toda la secuencia de letras recién colocadas junto a las que ya había en el tablero para aplicar puntualmente los multiplicadores de casilla, sumar el valor base de cada ficha y añadir el bono de 50 puntos si se han usado todas las fichas de la mano. Este total se compara con la mejor puntuación registrada hasta el momento; si la supera, se actualiza la jugada óptima guardando el conjunto de letras y sus posiciones. Una vez procesadas todas las anclas en ambas orientaciones (horizontal y vertical), el algoritmo devuelve finalmente la colocación de mayor puntuación junto con su valor total.

4. Análisis de coste y complejidad

Parámetros clave:

- B: número de fichas del jugador (máximo 7).
- L: espacios libres contiguos a la izquierda/arriba de un ancla.
- R: espacios libres contiguos a la derecha/abajo de un ancla.

4.1 Búsqueda de anclas.

El algoritmo revisa cada casilla de un tablero fijo de 15×15 (225 posiciones). En cada casilla comprueba sus vecinos inmediatos. Esta operación es constante O(1) por casilla, por lo que la búsqueda completa tiene un coste total O(225), constante en la práctica, insignificante en comparación con el resto del algoritmo.

4.2 Construcción inicial de prefijos.

Por cada ancla encontrada, el algoritmo retrocede (hacia la izquierda o arriba según orientación) hasta encontrar otra ficha o borde del tablero. Este retroceso consume como máximo hasta 14 pasos, teniendo así un coste máximo de O(L), siendo $L \leq 15$. Solo se busca la primera palabra valida de la primera ancla, en caso de no encontrar ninguna palabra valida, se pasa a la siguiente. Por tanto el peor caso posible, en el que no se encuentra ninguna palabra seria revisar todas las anclas, con un coste de L de media.

4.3 Generación de prefijos izquierdos (Before Part).

La generación de prefijos usa backtracking sobre el DAWG, considerando las fichas del jugador y las posiciones libres antes del ancla. Aunque teóricamente este proceso podría ser O(B^L), con $B \leq 7$ y $L \leq 7$, la estructura del DAWG poda rápidamente cualquier

prefijo que no pueda dar lugar a palabras válidas. Esto reduce enormemente la búsqueda, resultando en una exploración muy eficiente en la práctica. Esta etapa utiliza implícitamente una estrategia de branch & bound, cortando de inmediato cualquier camino que no pueda conducir a una palabra válida.

4.4 Extensión hacia la derecha y validación (Extend After).

Cada prefijo generado se extiende letra por letra hacia la derecha o abajo. Cada letra se valida inmediatamente mediante el DAWG, considerando dos aspectos: la disponibilidad de fichas en mano y la validez de palabras cruzadas formadas en perpendicular. Aunque la complejidad teórica podría ser O(B^R), con $R \leq 15$, las podas por validación de cruces y prefijos inválidos limitan enormemente la búsqueda real. Esta fase también utiliza branch & bound, ya que descarta rápidamente combinaciones inviables. Esto hace que la extensión sea rápida en términos prácticos, aunque menos eficiente que el uso de bit-vectors precomputados, ofreciendo a cambio mayor flexibilidad.

4.5 Cálculo de puntuación y selección final.

Cada palabra válida formada se evalúa sumando puntos según las letras usadas y multiplicadores del tablero. Este cálculo es lineal respecto a la longitud de la palabra generada. Dado que solo se consideran unas pocas centenas de jugadas por turno, el coste total es lineal y muy pequeño comparado con las etapas anteriores.